Python Iteration, aka Loops

Main Points

1. Just what is iteration anyway?
2. Definite iteration
3. Create lists of values using list comprehensions
4. Increment and decrement variables
5. Indefinite iteration
6. To infinity (or less)
7. Quitting on a loop

1 What is Iteration?

Iteration is the repetition of a segment of code a number of times. Sometimes the number of times that the code needs to repeat is known and sometimes it is not.

- Definite iteration is when the number of iterations is known ahead of time
- Indefinite iteration is when the number of iterations is not know ahead of time
- Typically (but not always)...
  - Definite iteration will be performed with `for` loops
  - Indefinite iteration will be performed with `while` loops
- Each iteration (or pass) through the code is often called a *loop*
- The act of iterating is often referred to as *looping*

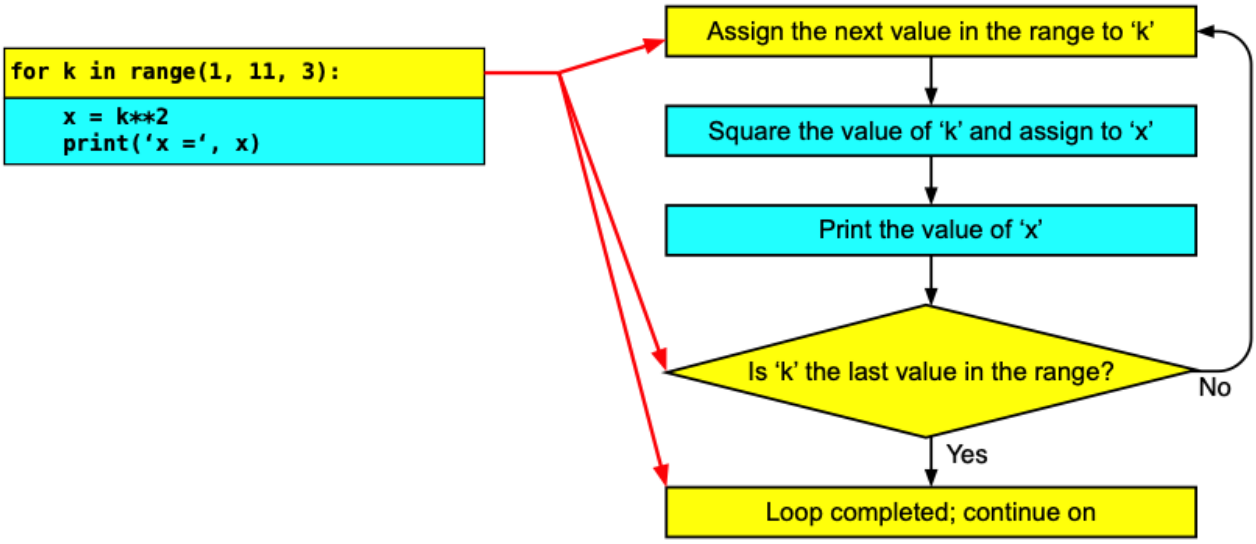
2 Definite Iteration with `for` Loops

Definite iteration means that a loop will iterate a specific number of times that are known before starting the loop. The best tool for definite iteration is the `for` loop. A `for` consists of a group/block of commands that is repeated a predetermined number of times.

- `for` loops requires the use of a sequence or iterator, such as...
  - Strings
  - Lists
  - Tuples
  - Ranges
  - Zip objects
- The following code block and illustration that follow illustrate the general structure of a `for` loop

General structure of a `for` loop

```
 1     2     3     4    5
for var_name in sequence:
    # commands/instructions to perform during each iteration
    continue  # remove this line, it is a placeholder
```



- The first line is the *header* which must have each of the following in the order shown (numbers match those in the code block above)
  1. The `for` command
  2. One (or more) iteration variable
  3. An `in` statement
  4. A sequence or iterator
  5. A colon
- The header line's indentation must match the current indentation level
- Remaining lines are the *body*
  - Must be indented 4 spaces relative to the header line
  - Includes the code to be executed each iteration
  - Must contain at least one command
  - The `continue` command is a good placeholder when initially writing a loop
- Each time through the loop...
  - Iteration variable is assigned the value of the first item in the sequence
  - Indented lines are executed using the assigned value of the iteration variable
- Execution continues with the next line after the loop once the values in the sequence are exhausted

Example `for` loop using `range()`

```
for k in range(1, 11, 3):
    x = k**2
    print('x =', x)
print('The loop has finished')
```

```
x = 1
x = 16
x = 49
x = 100
The loop has finished
```

Python Tutor `https://pythontutor.com/visualize.html#mode=edit` is a helpful visualization tool for more complex code. Copy the code from a code block into the provided code editing cell in *Python Tutor* then click on the "Visualize Execution" button.

The previous example could have used a list instead of a range, although a range is considered a better practice in this case. The example below demonstrates use of a list instead of a range that will produce the same results.

Example `for` loop using a list

```
for k in [1, 4, 7, 10]:
    x = k**2
    print('x =', x)
print('The loop has finished')
```

It is not necessary to actually use the iteration variable anywhere in the loop body, nor does the loop need to perform any math. An iteration variable still needs to be included one in the header though. The following code block demonstrates such a case. Notice that the sequence in the header is also a name that has been assigned a range beforehand.

Example of a `for` loop that does reference the iteration variable

```
good_list = range(4)
for nice_number in good_list:
    print("Spam")
```

```
Spam
Spam
Spam
Spam
```

2.1 Filling Lists Using `for` Loops

You can use `for` loops to fill lists using calculated values from the loop. Doing so requires that a list (usually empty) be created before the loop starts and then gets filled one item at a time as the loop runs.

- Creating two lists for making an x, y-plot (graph) is a good example
  - One list for the independent variable; i.e. x
  - Another for the dependent variable; i.e. y
  - An empty list is created for the dependent variable before the loop header line
  - The independent variable is used as the sequence in the header
  - The dependent variable value is calculated and appended to its list each pass
- The example in following code block...
  - Creates an empty list named `y` before the loop header line
  - Uses `x` for the iteration variable over a range of values
  - Calculates x^2 each iteration
  - Appends the calculated value to the list named `y`

Fill a list with calculated values using a `for` loop

```
y = []
for x in range(1, 11, 3):
    y.append(x**2)
    print('x =', x)   # print the current value of x each pass
    print('y =', y)   # watch the progress of the list being filled
print('\nFinal y =', y)
```

```
x = 1
y = [1]
x = 4
y = [1, 16]
x = 7
y = [1, 16, 49]
x = 10
y = [1, 16, 49, 100]

Final y = [1, 16, 49, 100]
```

The print statements in the above example are not normally required. They were included here to demonstrate the progress of the loop as it executed.

2.2 Using Existing Lists in `for` Loops

Values from an existing named list can be used as the sequence in a `for` loop. Use the list name in the sequence part of the `for` statement. Values from multiple lists can be used in a loop if each are the same size.

Below are three techniques for using the values from multiple lists in a loop.

1. Indexing each list
   - Use a range in the `for` statement
   - The range length must match the number of items in each list
   - The iteration variable will be the index used to access values from the lists
   - Example: if the iteration variable is `n`, use `list_1[n]` and `list_2[n]` in the loop
2. Using `enumerate()` and indexing together
   - Use a header like this: `for n, value in enumerate(my_list):`
   - `n` will be assigned the current index position in the list
   - `value` will be assigned the value from `my_list` at index `n`
   - Use the the index variable to access values from other lists via indexing
3. Using `zip()`
   - A more *Pythonic* approach
   - Zip multiple lists together in the `for` statement
   - For example: `for x, y, z in zip(list_x, list_y, list_z):`
   - Directly use the iteration variables without needing to use any indexing in the loop
4. Use a list comprehension (covered in a separate section)

2.2.1 Technique 1 - Indexing

`for` loop using mulitple lists – indexing

```
list1 = [1, 2, 3, 4, 5]
list2 = [5, 4, 3, 2, 1]
list3 = []
for index in range(len(list1)):
  list3.append(list1[index] * list2[index])

print(f"list1 + list 2 equals {list3}")
```

```
list1 + list 2 equals [5, 8, 9, 8, 5]
```

2.2.2 Technique 2 - Using `enumerate()` and Indexing

`for` loop using mulitple lists – `enumerate` and indexing

```
list1 = [1, 2, 3, 4, 5]
list2 = [5, 4, 3, 2, 1]
list3 = []
for index, list1_value in enumerate(list1):
    list3.append(list1_value * list2[index])

print(f"list1 + list 2 equals {list3}")
```

2.2.3 Technique 3 - using `zip()`

When using `zip()` in a `for` loop, the number of iteration variables needs to match the number of lists that were zipped together. The iteration variables need to be separated by commas and can either be within parentheses or not, for instance, both `for (x, y, z) in zip(a, b, c):` or `for i, j, k in zip(X, Y, Z):` are valid.

`for` loop using mulitple lists – using `zip()`

```
list1 = [1, 2, 3, 4, 5]
list2 = [5, 4, 3, 2, 1]
list3 = []
for x, y in zip(list1, list2):
    list3.append(x * y)

print(f"list1 + list 2 equals {list3}")
```

2.3 Nested `for` Loops

Placing one loop within another loop is referred to as nesting. Nesting loops is helpful when building lists of lists.

- The outer loop is started using its first iteration value
- The inner loop then iterates completely though its sequence
- Each time the inner loop finishes, the outer loop's iteration variable is assigned the next value
- The inner loop again runs completely
- The process continues until the outer loop's sequence is exhausted
- All inner loop code must be indented relative to the outer loop by 4 spaces
- Nesting can be any practical depth

The following nested loop code creates a list of lists such that the outer loop (iteration variable named `outer`) iterates over a range of integers from `1` to `3` (inclusive) and the inner loop (iteration variable named `inner`) iterates over a range of integers from `1` to `5` (inclusive). It appends the value `outer * inner` to each position of each of the sub-lists within the list named `C`.

Nested `for` loops to create a list of lists

```
C = []
for index, outer in enumerate(range(1, 4)):
    C.append([])  # add an empty list at the end of C
    for inner in range(1, 6):
        C[index].append(outer * inner)  # add to the latest sub-list
print(C)
```

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15]]
```

2.4 Iterating Over Strings

Strings are sequences of characters that can to be used as iteration sequences. Iteration over a string occurs one character at a time.

The following code block demonstrates iterating over the string `name`. Each pass a character is printed in uppercase. Note the keyword argument `end` that is included in `print()`. This argument changes the default behavior of `print()` in which a newline character `\n` is automatically added at the end of the printed line. In this case `***` is added instead of a newline character.

Iterating over a string

```
name = "Ferris"
for letter in name:
    print(letter.upper(), end=' *** ')
```

```
F *** E *** R *** R *** I *** S ***
```

3 List Comprehension; A Very *Pythonic* Technique

Python includes a technique referred to as a *list comprehension* that one might consider to be very *Pythonic*. It combines list creation, an expression, and a `for` statement into one command. The basic structure of a list comprehension is shown below.

```
variable = [<expression> for <var> in <sequence>]
variable = [<expression> for <var> in <sequence> if <comparison>]
```

- Must be placed in square brackets (not parentheses), hence the name "list comprehension"
- The expression should use the iteration variable in most cases
- Can include an optional `if` statement after the sequence
- The iteration variable is local to the list comprehension
- It is a good method to calculate values for plotting (used in another document)
- `y = [10*x for x in range(1, 6)]` assigns the list `[10, 20, 30, 40, 50]` to `y`
- List comprehensions can be nested similar to `for` loops

The following list comprehension does essentially the same thing as an earlier loop example did. List comprehensions do not generally contain `print()` commands.

Basic list comprehension example

```
y = [x**2 for x in range(1, 11, 3)]
print(y)
```

```
[1, 16, 49, 100]
```

The next example creates a formatted string for each value of `x`. This is not standard, but it may prove useful somewhere.

List comprehension example with formatted strings

```
y = [f"x = {x}, y = {x**2}" for x in range(1, 11, 3)]
print(y)
```

```
['x = 1, y = 1', 'x = 4, y = 16', 'x = 7, y = 49', 'x = 10, y = 100']
```

Printing within a list comprehension is not usually done, but it could be used to create a table of sorts. If `print()` is used in a list comprehension, the list that is created will contain a number of `None` objects, as can be seen in the following example.

Basic list comprehension example with `print()` statements

```
print("|  x  | x**2 |")
print("+-----+------+")
y = [print(f"| {x:2d}  |  {x**2:3d} |") for x in range(1, 11, 3)]
print()
print(y)
```

```
|  x  | x**2 |
+-----+------+
|  1  |    1 |
|  4  |   16 |
|  7  |   49 |
| 10  |  100 |

[None, None, None, None]
```

The following example demonstrates that the iteration variable within the list comprehension is not available outside of the list comprehension.

Demonstrating the local nature of the list comprehension iteration variable

```
# x in the comprehension is local and not available afterwards
x = 8675309
y = [10*x for x in range(1, 6)]
print("x =", x)   # The original x, not from the comprehesion
print("y =", y)
```

```
x = 8675309
y = [10, 20, 30, 40, 50]
```

The next example assigns a range to a variable before the list comprehension and uses that variable as the sequence in the list comprehension. Here the same name is used for the iteration variable as the sequence without any issues due to the iteration varaible being local to the list comprehension.

List comprehension using a previously defined sequence

```
import math
x = range(1, 11)
y = [round(math.sqrt(x), 3) for x in x]   # notice the for x in x
print(f"x = {x}")
print(f"y = {y}")
```

```
x = range(1, 11)
y = [1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.646, 2.828, 3.0, 3.162]
```

In the code block below nested list comprehensions are used to perform the same task as nested loops were previously used.

Nested list comprehension

```
C = [[outer * inner for inner in range(1, 6)]
     for outer in range(1, 4)]
print(C)
```

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15]]
```

List comprehensions can include an optional conditional statement. The conditional statement must be placed after the `for` statement. For instance, the following list comprehension will only perform a calculation when values in the sequence are less than 10.

List comprehension with optional conditional statement

```
z = [2*x for x in [1, 2, 5, 7, 10, 12, 8, 9, 15] if x < 10]
print(z)
```

```
[2, 4, 10, 14, 16, 18]
```

Why does the list generated in the following code block have four values when the iterator contains 50? *Answer*: they are the only values that are divisible by both 3 and 4 between 1 and 50 (inclusive). Note that the expression `(not x % 3)` will be `True` when the remainder of x $\div$ 3 is zero; meaning that `x` is divisible by 3.

Another list comprehension with conditions

```
y = [x for x in range(1, 51) if (not x % 3) and (not x % 4)]
print(y)
```

```
[12, 24, 36, 48]
```

A single list comprehension can create a list of lists or tuples by enclosing multiple expressions separated by commas inside square brackets or parentheses. The result will be a list of lists or tuples. Using `zip()` on the result can create two or more separate sequences of values by placing a `*` inside the opening parenthesis of the `zip()` function. This causes the list of lists/tuples (or a zip object) to be separated into individual tuples. Placing multiple variable names separated by a comma to the left of the equal sign tells *Python* to assign the separated tuples to those variable names. The following example demonstrates this interesting phenomenon.

List comprehension with unzipping

```
list1 = [[x, x**2] for x in range(1, 11, 3)]  # create list of lists
list2 = [(x, x**2) for x in range(1, 11, 3)]  # create list of tuples
tuple1, tuple2 = zip(*list1)
print(list1)
print(list2)
print(tuple1)
print(tuple2)
```

```
[[1, 1], [4, 16], [7, 49], [10, 100]]
[(1, 1), (4, 16), (7, 49), (10, 100)]
(1, 4, 7, 10)
(1, 16, 49, 100)
```

4 Getting Ready for `while` Loops: Increment Values

There is no need to manually update (increment) the iteration variable when using a `for` loop since the variable is automatically updated to the next value in the sequence each iteration. In general, incrementing means changing the value of a variable and assigning the new value to the original name; often the operation is addition , hence the name.

- Using `while` loops very often requires incrementing a variable
- `x = x + 1` is a common method to increment `x` by 1
  - Mathematically this appears to makes no sense, but programmatically it does
  - The equal sign is not used as an equality in *Python*; it is an assignment operator
  - The expression states that 1 is to be added to the current value of `x` and assigned back to `x`
  - The right side of the equal sign is completely evaluated before the assignment takes place

- Variables can be "incremented" by using...
  - Addition (aka incrementing)
  - Subtraction (aka decrementing)
  - Multiplication
  - Division
  - Exponentiation
- A more *Pythonic* way to increment is to use one of the shortcuts below

Traditional Expression	Shortcut Expression	Starting `x`	Ending `x`
`x = x + 1`	`x += 1`	3	4
`x = x - 1`	`x -= 1`	4	3
`x = x * 2`	`x *= 2`	3	6
`x = x / 2`	`x /= 2`	6	3
`x = x**2`	`x **= 2`	3	9

Example of incrementing a variable

```
>>> a = 5
>>> print(a)
5
>>> a += 4   # add 4 to a and assign back to a
>>> print(a)
9
>>> a -= 4   # subtract 4 from a and assign back to a
>>> print(a)
5
>>> a **= 2  # square a and assign back to a
>>> print(a)
25
>>> a *= 3   # multiply a by 3 and assign back to a
>>> print(a)
75
```

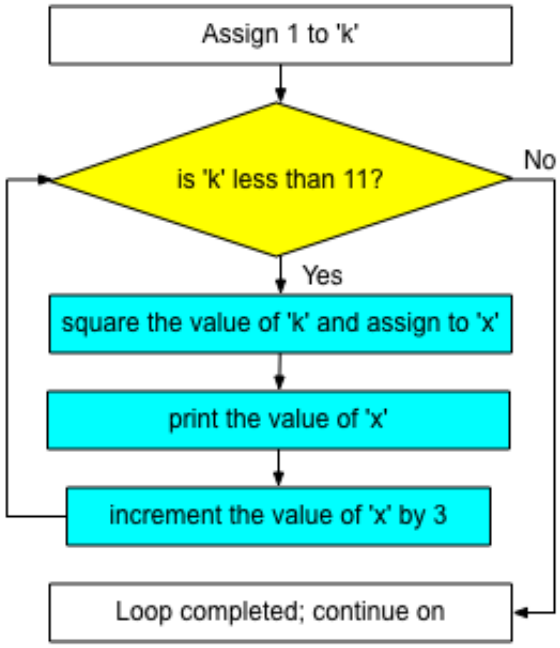
5 Indefinite Iteration with `while` Loops

A `while` loop is a group of commands that is repeated until a condition is met or there is a `break` command. The following code block and flow chart illustrate the general structure of a `while` loop.

General structure of a `while` loop

```
var = value
while condition:   # condition often involves var
    # commands/instructions to perform during each iteration
    # increment var (usually)
```

```
k = 1
while k < 11:
    x = k**2
    print('x =', x)
    k += 3
```



- Any variables used in the conditional statement need to be set to values before the loop starts
- First line of the group is the *header*
  - Starts with the `while` statement
  - Requires a conditional statement after the `while`
  - End the line with a colon
  - The header line's indentation must match the current indentation level
- Remaining lines are referred to as the *body*
  - Indented by 4 spaces relative to the header line
  - Includes the code to execute if the condition in the header is `True`
  - Will keep looping until...
    - The conditional statement in the header becomes `False`
    - A `break` command is executed
- The conditional in the header is checked before anything in the body is executed
  - If `True`, the code in the body run and program flow will return to the header
  - If `False`, the body will be skipped and the loop with end
- Usually a variable in the body will increment, causing the conditional to eventually become `False`
- The increment expression is usually done near the last line of the body

Example `while` loop with incrementing

```
k = 1
while k < 11:  # will not start another iteration if k >= 11
    x = k**2
    print('x =', x)
    k += 3     # without this line, the loop would never exit
```

```
x = 1
x = 16
x = 49
x = 100
```

The loop in the following code block uses the value of `x` that will be calculated in the body in the conditional statement. For this to work, `x` needs to initially be assigned a value that lets the loop run at least one iteration. The loop will execute another iteration as long as the last calculated value of `x` is less than 100, so the initial value is set to zero. It will stop looping once the last calculated value of `x` is greater than or equal to 100.

Example `while` loop with condition based on the calculated value

```
x = 0
k = 1
while x < 100:
    x = k**2
    print('x =', x)
    k += 3
```

```
x = 1
x = 16
x = 49
x = 100
```

In the following example the variable used in the conditional is halved each iteration using a division "increment." The loop will stop once the newest halved value is less than or equal to 1.

Halving a value each iteration of a `while()` loop

```
x = 100
while x > 1:
    x /= 2
    print(x)
```

```
50.0
25.0
12.5
6.25
3.125
1.5625
0.78125
```

The following example uses the last value in a list in the conditional statement. In a case like this, the list does not start empty, it starts with the value 1 in it before the loop header line. The rightmost value in the list (the -1 index position) is doubled each iteration. The loop will not execute again if the next calculated value would be greater than 1000.

Example `while` loop for appending values to a list

```
y = [1]
while y[-1] * 2 <= 1000:   # stop if the next value will be > 1000
    y.append(y[-1] * 2)
print('Final y =', y)
```

```
Final y = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

The above code block "looks ahead" or predicts what the next value will be in the conditional statement. Another way to accomplish the same task is to check if the last item in the list is less than or equal to 1000 in the conditional and then check after the loop if the value ended up being greater than 1000. If it is greater than 1000 the `list.pop()` method can be used to remove it from the list. The code below demonstrates this technique. It is not as *Pythonic* as the previous example, but it works.

A less *Pythonic* version of the previous example

```
y = [1]
while y[-1] <= 1000:   # stop if the last list value is > 1000
    y.append(y[-1] * 2)
if y[-1] > 1000:
    y.pop()
print('Final y =', y)
```

```
Final y = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

6 Infinite Loops

Infinite loops have a bad reputation because it is assumed the program is out of control. However, it is sometimes desirable to have a loop to run "infinitely" (or until a physical reset).Scripts for micro-controllers will usually include infinite loops where most of the commands in the script should run until the unit is physically turned off or restarted.

- Using just `True` as the conditional statement will "force" an infinite loop
- Interrupt infinite loops with the CTRL-C key combination in *Python*
- Avoid uncontrolled infinite loops by...
    - Ensuring that something within in the body will change the result of the conditional statement
    - Including a another means to exit a infinite loop, like using the `break` command

An uncontrolled infinite loop

```
print("Like the Energizer Bunny, this loop keeps going...")
while True:
    print("and going...")
```

The following example will cause an infinite loop because there is no increment expression or other means to change the variable used in the conditional statement.

An unintended infinite loop – missing an increment expression

```
x = 0
while x < 10:
    print(x)
```

An incorrect conditional statement will cause an infinite loop even when the variable used is incremented each iteration. In this example `x` will always be less than or equal to 10 since it is decremented (reduced by one) each iteration. Changing the conditional statement to `x > 1` will fix this problem.

An unintended infinite loop – improper conditional statement

```
print("Count down to launch...")
x = 10
while x <= 10:
    print(x, end="...")
    x -= 1
print("Take off")
```

7 The `break` Command

Using `while True:` may eliminate the need for incrementing a variable, but another means will be required to eventually stop the loop. Since `True` is *always* `True` the conditional result in the header line will never change. The `break` command is designed for stopping the execution of a loop.

- A `break` command along with an `if` statement can be added in the loop to force an exit
- If the `break` condition is not met, the loop will continue running
- The `break` command can also be used within a `for` loop to force it to exit early
- A `break` will only exit the current loop; multiple `break` commands may be needed for nested loops

A `while` loop and a `break` command have been added to the `diceroll()` function from a previous lecture. An `input()` function was added after the dice roll results are displayed requesting the user to enter `q` key to quit. The loop breaks if the first character of the user input value is either a `q` or `Q`.

Example function with `while` and `break`

```
def diceroll():
    import random
    while True:
        die1 = random.randint(1,6)
        die2 = random.randint(1,6)
        print('Die 1 =',die1,'    Die 2 =',die2)
        if die1 + die2 == 2:
            print(f'{die1} + {die2} is snake eyes')
            print('Too bad')
        elif (die1 + die2 == 7) or (die1 + die2 == 11):
            print(f'{die1} + {die2} = {die1 + die2}, a natural')
            print('Winner, winner, chicken dinner')
        elif die1 + die2 == 12:
            print(f'{die1} + {die2} is boxcars')
            print("It's not your day")
        else:
            print(f'{die1} + {die2} is nothing special')
            print('Better luck next time')
        quit = input("Enter q to quit: ")
        if quit.lower().startswith('q'):
            break
        print()
```

```
>> diceroll()
Die 1 = 1     Die 2 = 2
1 + 2 is nothing special
Better luck next time
Enter q to quit:

Die 1 = 4     Die 2 = 3
4 + 3 = 7, a natural
Winner, winner, chicken dinner
Enter q to quit:

Die 1 = 1     Die 2 = 1
1 + 1 is snake eyes
Too bad
Enter q to quit: quit
>>>
```

8 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapter 7
- *The Coder's Apprentice*, Pieter Spronck, chapter 7
- *A Practical Introduction to Python Programming*, Brian Heinold, chapters 2 and 9
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapters 6 and 7